Lecture 7: More Haskell

CSCI 101
Spring, 2019

Kim Bruce

Last Time:

- Closure properties of regular languages
  - Including pumping lemma
- Haskell programming

Pattern Matching

- Decompose lists:
  - [1,2,3] = 1:(2:(3:[]))
- Define functions by cases using pattern matching:

```
prod [] = 1
prod (fst:rest) = fst * (prod rest)
```

More Pattern Matching

- (x,y) = (5 `div` 2, 5 `mod` 2)
- hd:tl = [1,2,3]
- hd:_ = [4,5,6]
  - "_" is wildcard.

Static Typing

- Strongly typed via type inference
  - head:: [a] → a
    tail:: [a] → [a]
  - last [x] = x
    last (hd:tail) = last tail
- System deduces most general type, [a] -> a

Local Declarations

```
roots (a,b,c) =
   let      -- indenting is significant
      disc = sqrt(b*b-4.0*a*c)
   in
      ((-b + disc)/(2.0*a),(-b - disc)/(2.0*a))

*Main> roots(1,5,6)
(-2.0,-3.0)
```

or

```
roots' (a,b,c) = ((-b + disc)/(2.0*a),
                  (-b - disc)/(2.0*a))
   where disc = sqrt(b*b-4.0*a*c)
```

Anonymous functions

- dble x = x + x
- *abbreviates*
- dble = \x -> x + x

Defining New Types

- Type abbreviations
  - type Point = (Integer, Integer)
  - type Pair a = (a,a)
- data definitions
  - create new type with constructors as tags.
  - generative
- data Color = Red | Green | Blue

  See more complex examples later

Type Classes Intro

- Specify an interface:
  - class Eq a where
    (==) :: a -> a -> Bool -- specify ops
    (/=) :: a -> a -> Bool
    x == y = not (x /= y) -- optional implementations
    x /= y = not (x == y)
  - data TrafficLight = Red | Yellow | Green
    instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False

Common Type Classes

- Eq, Ord, Enum, Bounded, Show, Read
  - See http://www.haskell.org/tutorial/stdclasses.html
- data defs pick up default if add to class:
  - data ... deriving (Show, Eq)
- Can redefine:
  - instance Show TrafficLight where
    show Red = "Red light"
    show Yellow = "Yellow light"
    show Green = "Green light"

More Type Classes

- class (Eq a) => Num a where ...
  - instance of Num a must be Eq a
- Polymorphic function types can be prefixed w/ type classes
  - test x y = x < y *has type* (Ord a) => a -> a -> Bool
  - *Can be used w/ x, y of any Ord type.*
- *More later ...*
  - *Error messages often refer to actual parameter needing to be instance of a class -- to have an operation.*

Higher-Order Functions

- Functions that take function as parameter
  - Ex: map:: (a → b) → ([a] → [b])
- Build new control structures
  - listify oper identity [] = identity
    listify oper identity (fst:rest) =
    oper fst (listify oper identity rest)
  - sum' = listify (+) 0
    mult' = listify (*) 1
    and' = listify (&&) True
    or' = listify (||) False

Exercise

- Is listify left or right associative?
  - What is listify (-) 0 [3,2,1]? 2 or -6 or 0 or ???
- How can we change definition to associate the other way?

See built-in foldl and foldr

Quicksort

```
partition (pivot, []) = ([],[])
partition (pivot, first : others) =
   let
      (smalls, bigs) = partition(pivot, others)
   in
      if first < pivot
         then (first:smalls, bigs)
         else (smalls, first:bigs)
```

Type is:

```
partition :: (Ord a) => (a, [a]) -> ([a], [a])
```

Quicksort

```
qsort [] = []
qsort [singleton] = [singleton]
qsort (first:rest) =
   let
      (smalls, bigs) = partition(first,rest)
   in
      qsort(smalls) ++ [first] ++ qsort(bigs)
```

Type is:

```
qsort :: (Ord t) => [t] -> [t]
```

Quicksort - parametrically

```
partition (pivot, []) lThan = ([],[])
partition (pivot, first : others) lThan =
  let
    (smalls, bigs) = partition(pivot, others) lThan
  in
    if (lThan first pivot)
      then (first:smalls, bigs)
      else (smalls, first:bigs)

partition ::
        (t, [a]) -> (a -> t -> Bool) -> ([a], [a])

*Main> partition(6,[8,4,6,3])(>)
```

Quicksort

```
qsort [] lt = []
qsort [singleton] lt = [singleton]
qsort (first:rest) lt =
   let
      (smalls, bigs) = partition (first,rest) lt
   in
      qsort smalls lt ++ [first]
                      ++ qsort bigs lt

qsort :: [a] -> (a -> a -> Bool) -> [a]

*Main> qsort [33,66,32,87,999,2](>)
[999,87,66,33,32,2]
```

Recursive Datatype Examples

- data IntTree = Leaf Integer |
  Interior (IntTree,IntTree)
  deriving Show
  - Example values: Leaf 3, Interior(Leaf 4,Leaf -5), ...
- data Tree a = Niltree |
  Maketree (a, Tree a, Tree a)